April 2019

Collision Identification Program (CIP)

Krysta Rose Murdy
Worcester Polytechnic Institute

Collision Identification Program (CIP)

A Collision Detection Toolkit for Point Clouds

A ***Major Qualifying Project*** Report
Submitted to the Faculty of the

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the
Degrees of Bachelor of Science in
Computer Science
and
Electrical & Computer Engineering
by

Krysta R Murdy

Project Advisors:

Professor Alexander M Wyglinski, ECE

Professor Hugh C Lauer, CS

25 April 2019

Abstract

The goal of this project is to develop a functional collision detection toolkit independent of hardware and additional software. The fundamental features of this project include the following: accessing and parsing the desired data, checking for significant changes in the point cloud, and determining basic information on the points where a movement is detected. In order to enhance practicality for collision detection, a collision flag able to be implemented in an external interrupt routine was included. Many of these features can be found in existing collision detection software, however, these products are often costly, require specific hardware, and are challenging to implement with an existing product. For this reason, the collision detection software created in this project, CIP, is a Toolkit API able to work with any software or hardware that processes point clouds. Going forward, this project could be continued in several ways, mainly: object identification and pathfinding for collision avoidance. Ultimately, CIP is designed as a collision detection Toolkit API for point clouds that is able to be offered to consumers and smaller organizations who may not have a need for the more costly and complicated existing options.

Acknowledgements

This project would not have been possible without...

- Guidance from my advisors, Professors Wyglinski and Lauer,
- Support and a LiDAR from the lovely people at HYPACK, and
- Encouragement from my family and friends, with nothing more inspiring than my grandfather's excitement!

Thank you all for your passion and support!!

Contents

Abstract i

Acknowledgements ii

Contents iv

List of Figures v

List of Tables vi

1 Introduction 1

2 Point Cloud & LiDAR Fundamentals 6

3 Proposed Approach 14

4 Implementation & Methodology 16

VLP-16 Driver Development 16

Creating the CIP Toolkit API 20

Forming the Complete Program 20

5 Results & Discussion 22

6 Conclusions & Future Work 25

References 26

Appendix A: Main Program CPP File 28

Appendix B: VLP-16 Driver CPP File 30

Appendix C: VLP-16 Driver Header File 38

Appendix D: CIP Toolkit API CPP File 41

Appendix E: CIP Toolkit API Header File 45

Appendix F: Global Data Settings Files 47

Appendix G: Winsock Tutorial Code **48**

Appendix I: VLP-16 Azimuth Pseudocode **52**

List of Figures

1 Results Graph for Self-Navigation Technological Race [3] 1
2 Results Table for Self-Navigation Technological Race [3] 2
3 Point Cloud Data Visualization Incorporating Other Traits [5] 3
4 Possible Ways to Visualize Point Cloud Data 4
5 VLP-16 Coordinate System Representation [10] 7
6 3D Point Cloud Data Visualization With VLP-16 8
7 3D Point Cloud Data Visualization Outside 8
8 3D Point Cloud Data Visualization Side View [11] 9
9 VLP-16, Velodyne Puck LiDAR Sensor [10] 9
10 Inside the VLP-16 [10] 10
11 Azimuth, or Angle Alpha, Put Into Perspective 11
12 VLP-16 Data Packet Structure [10] 12
13 Example of VLP-16 Raw Data, Start of Data Packet [10] 13
14 Example of VLP-16 Raw Data, End of Data Packet [10] 13
15 Project Milestone and Timeline Flowchart 15
16 Class Diagram for the Program 17
17 Structs Created for the Raw Data Packet 19
18 Implementing CIP in the Main Class 21
19 Results of a Code Analysis on the Program 24

List of Tables

1 Conversions from Spherical to Cartesian Coordinates [9] 6
2 Project Goals and Where They Are Implemented 15

1 Introduction

Self-navigation is gaining momentum as technological innovation has led to its inclusion in cars, drones, planes, robots, and numerous other applications. As modern computing technologies are increasingly powerful at cost effective prices, the demand for this method of easier, more computer-controlled movement has risen. This can be seen by the influx of companies striving to develop significantly more affordable self-driving cars [1]. This technological race is rapidly gaining attention and momentum as companies continue to push forward for the most self-sufficient and affordable driverless vehicle [2]. Dozens of companies are participating in this race, uncovering new knowledge and causing greater demand for innovation in related fields as well. The current results for this race are shown in Figure 1 [3]. As can be seen in this figure, this area of innovation has the attention of some very influential companies. Figure 2 shows these results in a table form and demonstrates how the success of commercialized self-navigation could lead to lower accidents and higher road efficiency by eliminating human error [3].

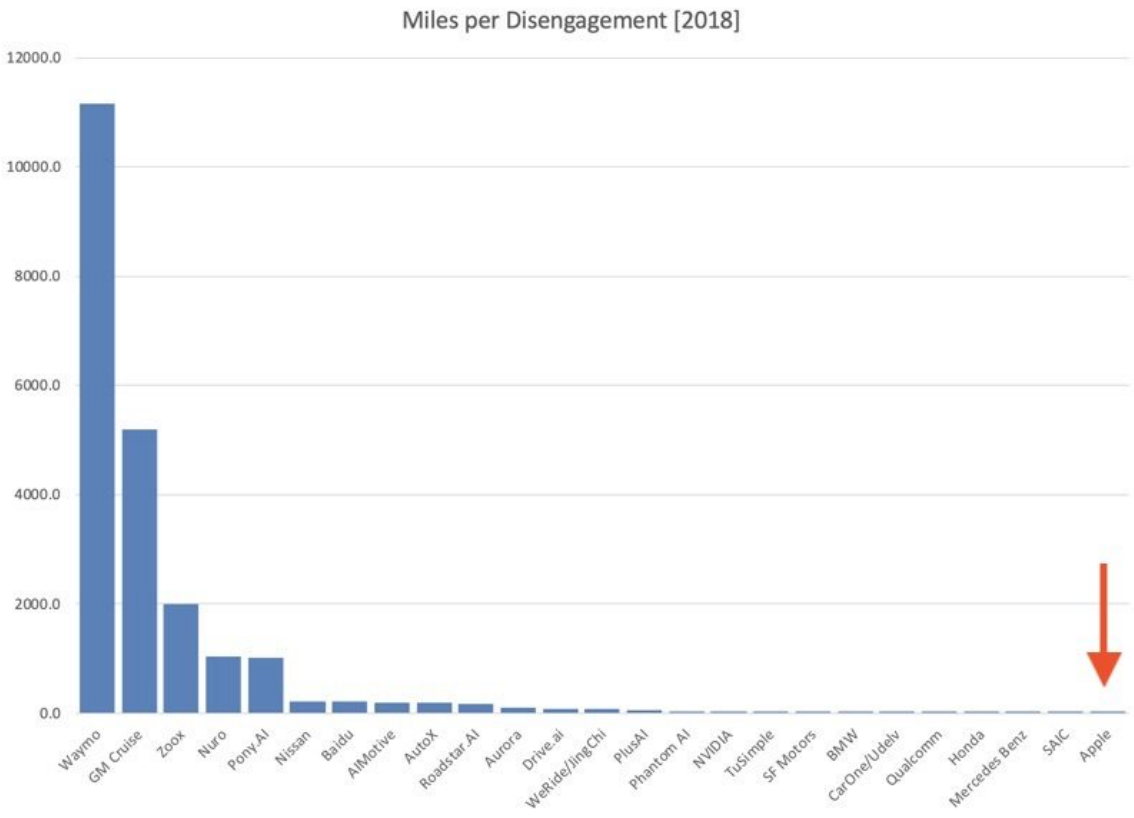


Figure 1: Results Graph for Self-Navigation Technological Race [3]

2018	Disengagements per 1000 miles	Miles per Disengagement	Disengagements per 1000 Kilometers	Kilometers per Disengagement
Waymo	0.09	11154.3	0.06	17846.8
GM Cruise	0.19	5204.9	0.12	8327.8
Zoox	0.50	2000.0	0.31	3200.0
Nuro	0.97	1028.3	0.61	1645.3
Pony.AI	0.98	1022.3	0.61	1635.6
Nissan	4.75	210.5	2.97	336.8
Baidu	4.86	205.6	3.04	329.0
AIMotive	4.96	201.6	3.10	322.6
AutoX	5.24	190.8	3.27	305.3
Roadstar.AI	5.70	175.3	3.56	280.5
Aurora	10.01	99.9	6.26	159.8
Drive.ai	11.91	83.9	7.45	134.3
WeRide/ JingChi	13.16	76.0	8.23	121.5
PlusAI	18.40	54.4	11.50	87.0
Phantom AI	48.20	20.7	30.13	33.2
NVIDIA	49.73	20.1	31.08	32.2
TuSimple	86.10	11.6	53.81	18.6
SF Motors	90.56	11.0	56.60	17.7
BMW	219.51	4.6	137.20	7.3
CarOne/ Udelv	260.27	3.8	162.67	6.1
Qualcomm	416.63	2.4	260.39	3.8
Honda	458.33	2.2	286.46	3.5
Mercedes Benz	682.52	1.5	426.58	2.3
SAIC	829.61	1.2	518.51	1.9
Apple	871.65	1.1	544.78	1.8

Figure 2: Results Table for Self-Navigation Technological Race [3]

Self-navigation requires the ability to assess situations quickly and completely in real-time, largely for collision aversion. For this to be possible, the product must be able to maintain a map of the surroundings, perform calculations and risk analysis on incoming data, and much more, possibly even having the ability to learn from past encounters. In order for all of this to take place, there must be a way to collect information about the world around the product, and this is accomplished through sensors. Thus, achieving self-navigation begins with efficiently processing large quantities of data from numerous sensors in order to infer the surrounding layout and current situation. A common form of receiving data for surrounding objects is through a data visualization format known as a point cloud. For the purpose of this project, a *point cloud* is a 3D view of a space represented as a multitude of "x," "y," "z" coordinates and other information marking object boundaries as observed from the sensor's point of view. Figure 4 shows some examples of how point clouds can be visualized, also demonstrating variation based on how data is gathered. Often the data held in this structure also includes other information regarding various properties of detected objects. Examples of this are the various colors in some of the visualizations in the figures below. The various colors could represent a material change, a specific distance or height, or numerous other information as well. This is very easy to see in Figure 3. Processing and interpreting this kind of information in real time would be invaluable for detecting and avoiding hazards and collisions [4].

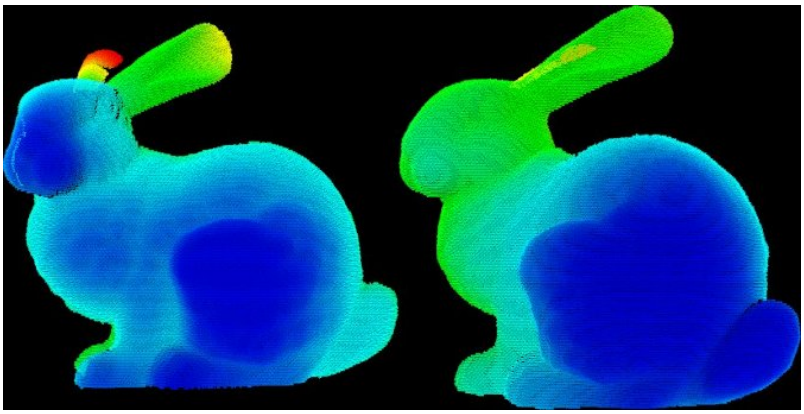
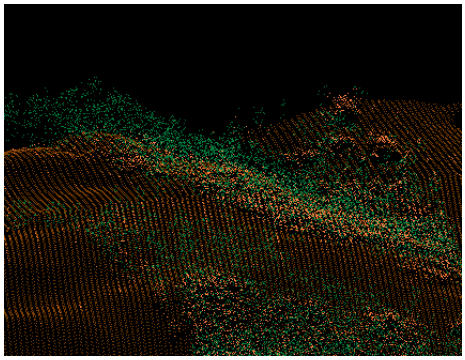
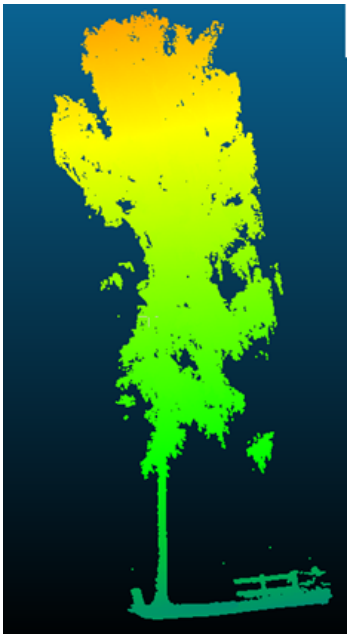


Figure 3: Point Cloud Data Visualization Incorporating Other Traits [5]



(a) Airborne Sensor [6]

(b) Grounded Sensor [7]



(c) Mobile Sensor [6]

Figure 4: Possible Ways to Visualize Point Cloud Data

With a the primary focus being on an efficient and accurate collision detection software, some aspects most appealing to potential customers, namely price, intuitiveness, and adaptability have been overshadowed [1]. Often these products are designed for specific hardware, tied to a full program, connected to a unique sensor-suite, or something along these lines. This yields a product bogged down by the price of hardware and additional software that is designed to work by itself.

With the current solutions, if an existing product, one with fully implemented hardware and software, gains the need for collision detection, the entire product would most likely need to be reworked. To implement one of the current collision detection solutions into a product such as this would require extensive rewriting of

the existing program and possibly also replacing a large portion of the hardware to accommodate for the restrictions of the collision detection program. This extra work is unnecessary and incredibly expensive.

With so much work and progress towards an efficient and accurate collision detection solution, the focus now must shift towards the needs of possible users: price, intuitiveness, and adaptability. The current need is for a collision detection product independent of specific hardware and programs; one that can be easily incorporated with an existing product. In order to achieve this, the program needs to be setup such that the collision detection processes are separate from any type of data processing and user interfacing. The goal is for effective simplicity.

Once an intuitive and adaptable solution for collision detection is created, even better solutions become a possibility, such as one capable of performing computations like the highly accurate and efficient versions yet with the simplicity of this one.

This report details the process of developing the basis for a point cloud collision detection toolkit that can work with point cloud producing sensors. The fundamental features of this project include the following: accessing and parsing the desired data, checking for significant changes in the point cloud, and determining basic information on the points where a movement is detected. In order to enhance practicality for collision detection, a collision flag able to be implemented in an external interrupt routine was included. This report walks through the initial approach to a solution, the process of implementing a design, the discussion of results and how the program evolved, and finally the possible future applications. Many of these features can be found in existing collision detection software, however, these products are often costly, require specific hardware, and are challenging to implement with an existing product. For this reason, the collision detection software created in this project was made into a Toolkit API able to work with any software or hardware that processes point clouds. Going forward, this project could be continued in several ways, mainly: functionality on a moving sensor, object identification, and determining possible paths to avoid a collision. Ultimately, the goal of this project is to create a collision detection toolkit for point clouds that is functional on a stationary sensor and could be marketed to consumers and smaller organizations who may not have a need for the more costly and complicated existing options.

2 Point Cloud & LiDAR Fundamentals

This section provides the basics on the point cloud data type and LiDARs. Included is a breakdown of the specific data packet produced by the LiDAR used for this project, and each of its aspects with their uses are explained. Upon completion of this chapter, the reader should have a comprehensive understanding of this data packet and how each of the individual pieces of information are used.

Developing a point cloud collision detection toolkit requires enough understanding of point cloud data to manipulate the data set for calculations. For this project, familiarity with point clouds was imperative for developing a comprehensive program for testing. A point cloud, in its simplest form, is a set of data points in a defined coordinate system.

The point clouds used for this project were a set of 3D data points in the Spherical coordinate system. The Spherical coordinate system denotes a position in 3D space as if the point is on the edge of a sphere by indicating an alpha angle, α, on the horizontal plane, an omega angle, ω, elevation on the vertical plane, and a radius, r [8]. In order to attain typical (x, y, z) coordinates, calculations need to be made to convert from Spherical to Cartesian coordinates. Spherical coordinates are a representation of points in a sphere within the (x, y, z) coordinate system, meaning that the alpha angle is in the xy-plane and the omega angle is a change down from the positive z-axis [8]. A visual representation of the Spherical coordinate system in general and relative to the sensor used is shown in Figure 5. See Table 1 below for the equations from Spherical to Cartesian coordinates.

Table 1: Conversions from Spherical to Cartesian Coordinates [9]

Cartesian	**Spherical**
x	r*sin(ω)cos(α)
y	r*sin(ω)sin(α)
z	r*cos(ω)

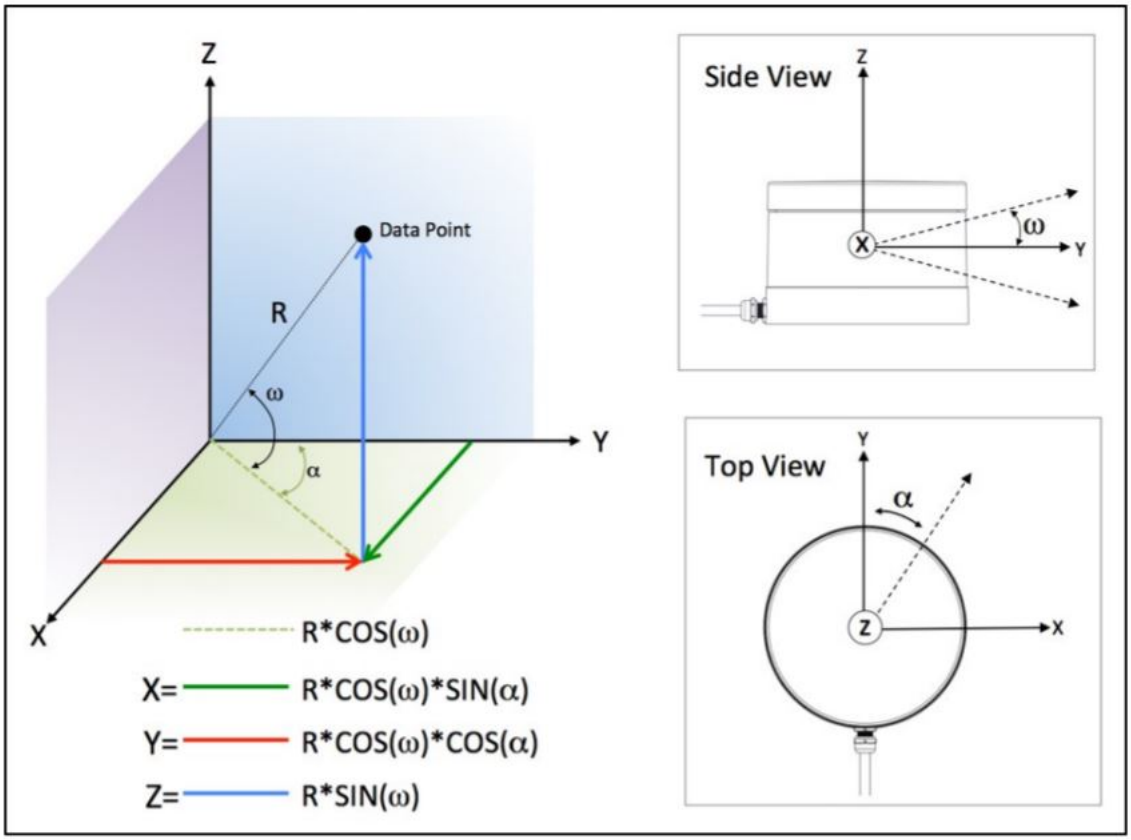


Figure 5: VLP-16 Coordinate System Representation [10]

Displaying all of the points in point cloud can be done using the right software. When displayed together, the points print and image of the surroundings as seen in Figure 6. In this image, the white dot at the top is the sensor that scanned the environment and created the point cloud. This sensor was on a drone above a park, and the resulting image shows the trees and ground with incredible detail, including a color gradient for different object heights. The next image, 7, is another image taken with the same sensor but from a couple meters above the ground. The sensor was positioned outside with a building on one side and thick shrubbery on the other. The windows of the building stand out as black squares because the light from the sensor was not reflected, so there are no points in those areas. The circular lines near the center of the image indicate the location of the sensor. In this case, the sensor was positioned in the very center of the circle, but, because it was connected to a GPS system, the point cloud can be seen shifting as the sensor moves. This is why some aspects of the image are doubled. The third point cloud image, Figure shows a top down of the sensor in a vertical orientation – so as if looking at the sensor's

side as shown in the **Side View** section of Figure 5. When compared to the image of how the sensors are positioned in the device, Figure 10, how the data is recorded can be easily observed by the parallel of the two figures.

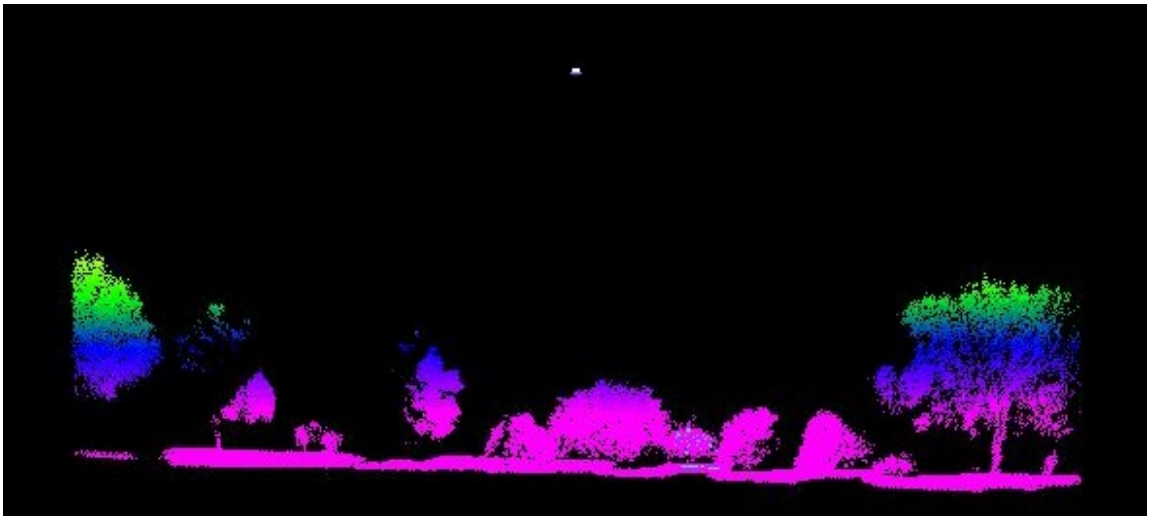


Figure 6: 3D Point Cloud Data Visualization With VLP-16

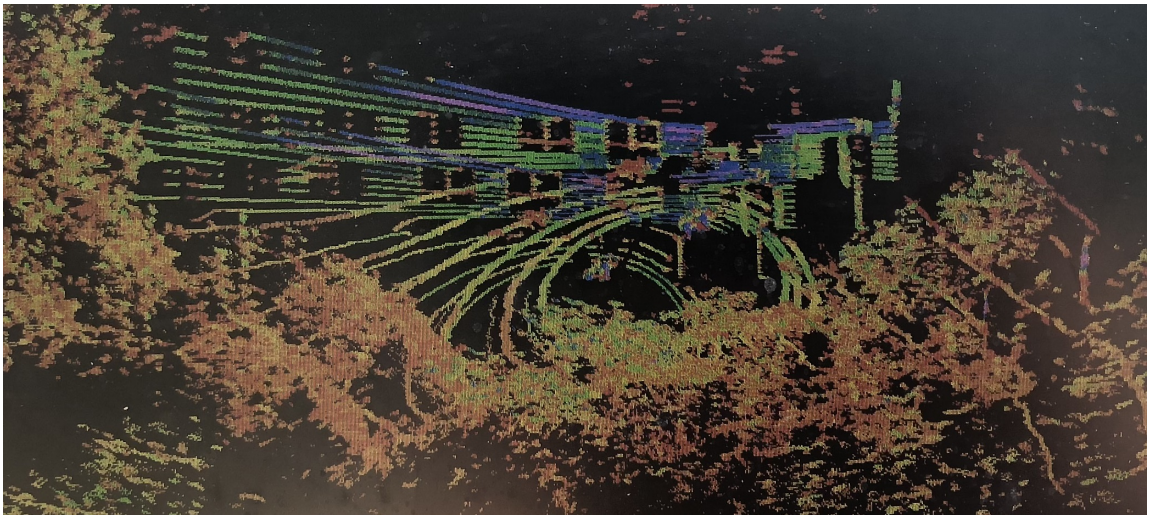


Figure 7: 3D Point Cloud Data Visualization Outside

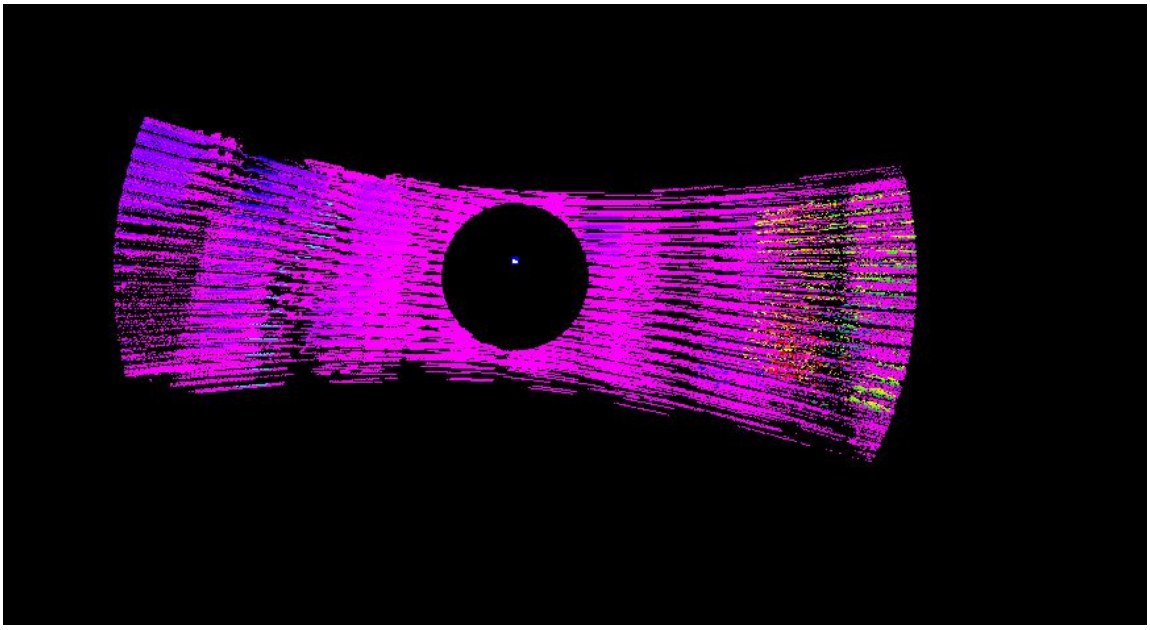


Figure 8: 3D Point Cloud Data Visualization Side View [11]

These figures were actually taken several months ago by the same sensor used in this project for obtaining point clouds for testing: a Velodyne Puck LiDAR, specifically the VLP-16. A LiDAR, short for Light Detection And Ranging, is a sensor that uses light in the style of radar. The VLP-16, has 16 infrared lasers, each with an infrared detector, that internally spin at a default of 600 RPM to scan the surroundings [10]. Every laser fires about 18,000 times per second, yielding nearly 300,000 data points every second when in single-return mode[1] [10]. The VLP-16 is shown below – inside and out, Figures 10 and 9 respectively.

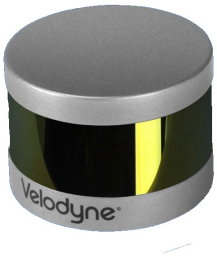


Figure 9: VLP-16, Velodyne Puck LiDAR Sensor [10]

[1]The VLP-16 has multiple options for interpreting and returning laser readings. In single-return mode each laser/detector pair returns one measurement per firing. This is the mode used in this project and greater detail is not necessary for understanding the project or this report.

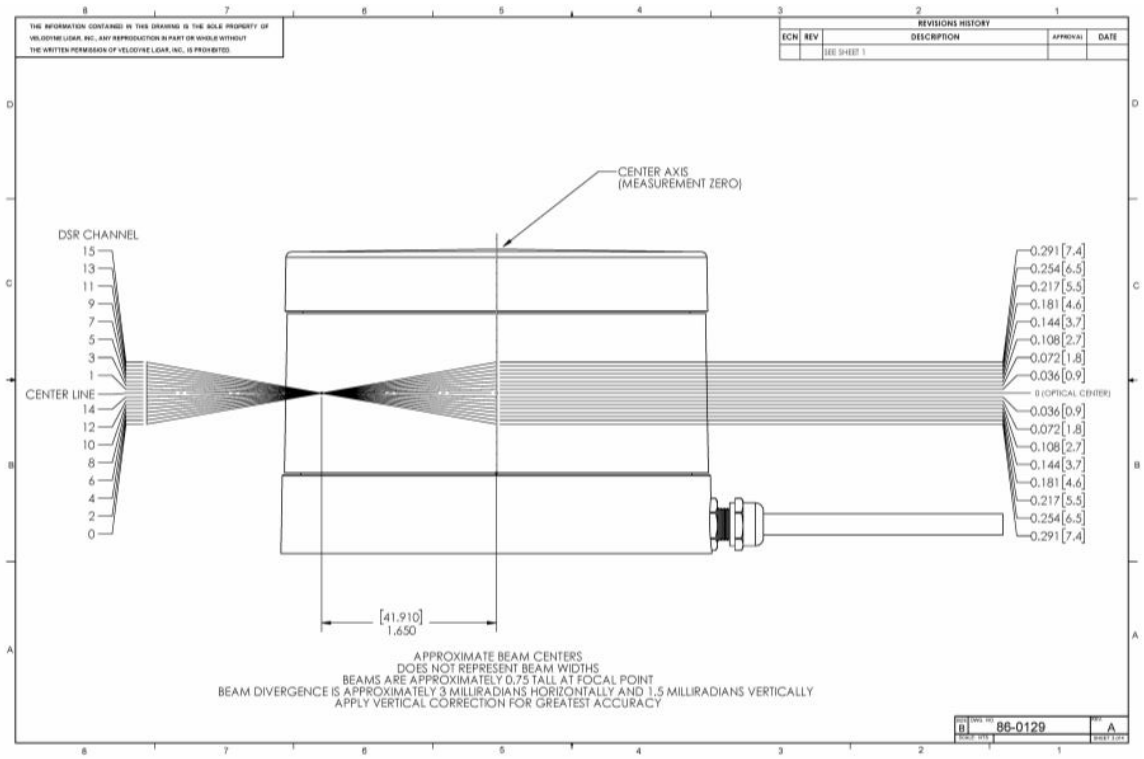


Figure 10: Inside the VLP-16 [10]

A bounty of data points is only one part of the extensive data packet returned by the VLP-16. The VLP-16 data packet is one instance of information from the sensor, and each one contains a protocol header[2], a timestamp, factory bytes, and data from 24 firing sequences contained in 12 data blocks. All of this results in 1,248 bytes containing enough information to explore a magnitude of implementations.

The timestamp contained in this data packet is a 32-bit unsigned integer that indicates the moment the first data point of the first firing sequence in the first data block is recorded. This timestamp represents the number of elapsed microseconds since the start of the current hour [10]. This value is particularly important for any georeferencing applications. In this project, the timestamp is primarily used when determining the velocity of moving objects.

The factory bytes indicate the device model with a product ID and the previously mentioned return mode. The device model determines the order of vertical angles used for calculating the omega angle sine and cosine offset. The return mode determines how the azimuths and data points are organized within the

[2]The protocol header is necessary for sending information via a UDP packet and is not used in any significant way for this project [10].

data packet [10].

A data block is where the necessary location information is located. A data packet has 24 data blocks, and each data block contains a flag[3], an azimuth, and two firing sequences.

An azimuth is the same as the alpha angle of the Spherical coordinate system. The azimuth value stored in a data block is an unsigned integer and represents the alpha angle in 100ths of a degree [10]. This angle is the location horizontally around the VLP-16 of the first laser in the first sequence at the time of firing. The azimuth is necessary for converting from Spherical to Cartesian coordinates which is necessary for locating the position of movement later on in this project. For a visualization of the azimuth, alpha angle, value, see Figure 11.

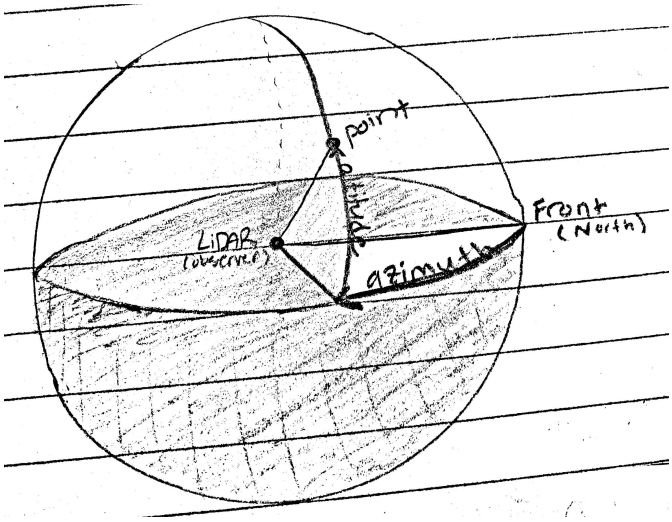


Figure 11: Azimuth, or Angle Alpha, Put Into Perspective

A firing sequence takes approximately 55.296 µs and is the firing and recharge of all 16 laser channels of the VLP-16. A laser channel is one 903 nm laser emitter and infrared detector pair. Each channel has a laser ID number and a fixed elevation angle relative to the sensor's horizontal plane – the omega angle in the Spherical coordinate system [10]. The omega angle for each laser is assigned based on the model of device and is determined by the laser's location in the data packet. This value is also necessary for conversions and determining locations in this project. Each laser channel produces a three byte data point thus resulting in 32 data points

[3]The flag is only used to determine the beginning of a data block and to ensure that the data is not stored improperly [10].

per data block and 384 per data packet [10]. Two of the data point bytes are for an unsigned integer representing the distance to the detected object within 2 mm granularity. The third byte represents calibrated surface reflectivity, which is an assigned value for the strength of the returned laser beam. The VLP-16 measures object reflectivity using comparisons with known targets in factory testing. The value is placed on a range from diffuse reflector to retroreflector [10]. The specificity of this value allows for potential applications in object identification.

A visual break down of the VLP-16 data packet is shown in Figure 12 below.

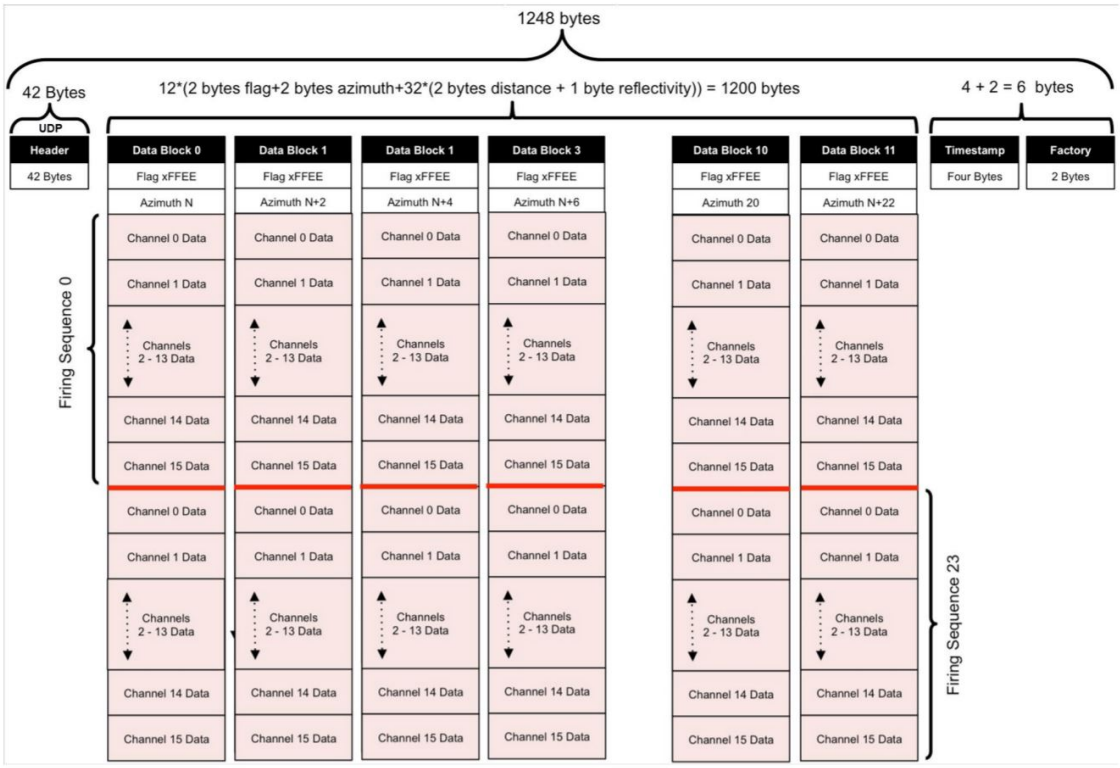


Figure 12: VLP-16 Data Packet Structure [10]

The VLP-16 data packet is a series of hexadecimal values that correspond to the different aspects of the data packet based on their order as previously discussed. Two examples of the raw data packet can be seen in Figure 13 which shows the start of the data packet with some calculations for the represented data and Figure 14 which shows the end of the data packet with some other calculations.

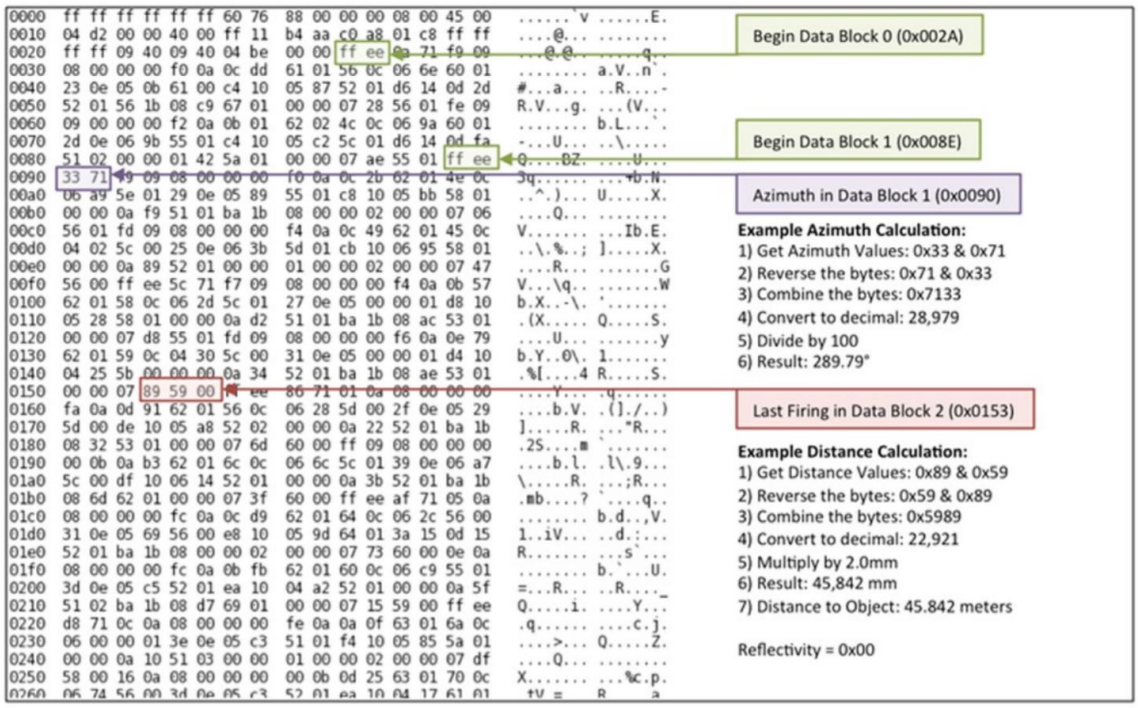


Figure 13: Example of VLP-16 Raw Data, Start of Data Packet [10]

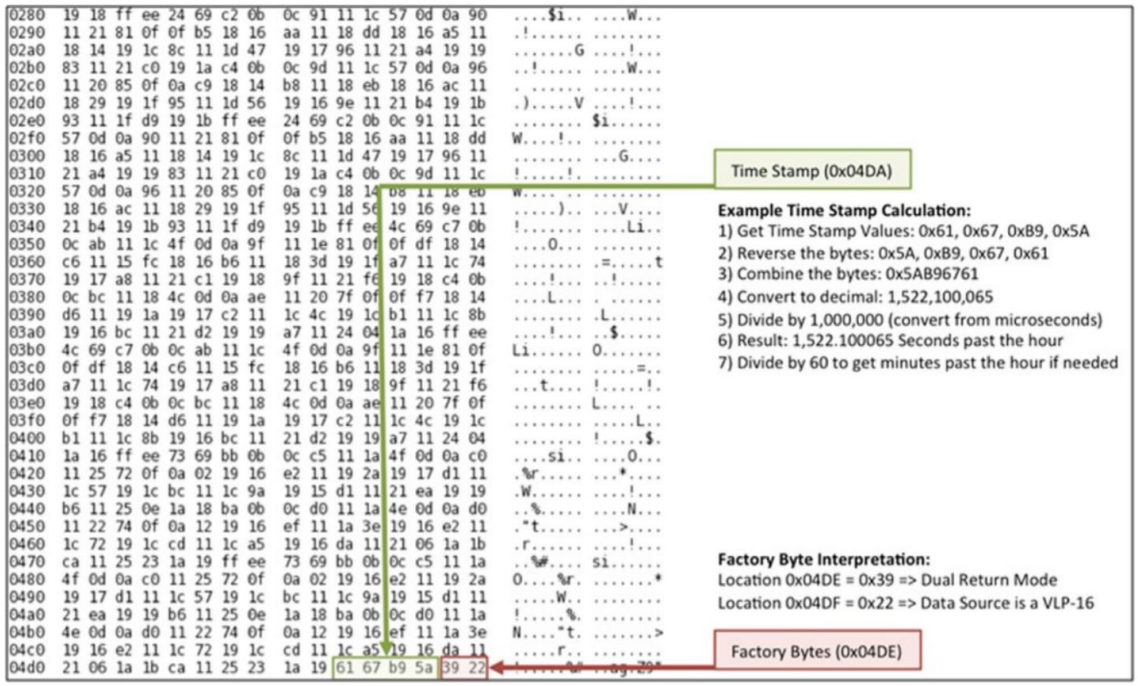


Figure 14: Example of VLP-16 Raw Data, End of Data Packet [10]

3 Proposed Approach

This section details the approach originally proposed for this project. The main goals of the project are overviewed and the final deliverable is detailed in order to give a complete view of the project.

The goal is to develop a functional collision detection toolkit independent of hardware and additional software. The Toolkit API assumes data collection and parsing has occurred, the corrected data is accessible, and any user collision notifications are handled elsewhere. Therefore, the Collision Identification Program, CIP for short, is primarily a collection of calculations and "getter" methods able to return desired collision information.

The basis of any collision detection is the evaluation of data to determine if any surrounding object moves towards the sensor. When a change is detected, the object's position and possible acceleration is determined in order to achieve accurate threat analysis. The logical next step is to relay this information somewhere so a possible collision does not go unnoticed. This can be achieved with a basic interrupt routine for collision alerts. These features are important in all collision detection, so the uniqueness of CIP lies in its design. In order to achieve the desired independence and modularity, the toolkit could be developed as an API. This method is ideal for simple, intuitive incorporation with existing code as it has the ability to output all important collision information without requiring anything beyond the users implementation.

In order to test functionality, CIP needs to be implemented into an existing program able to access, parse, and store point cloud data as well as call the API. For this reason, the creation of a device driver for the VLP-16 is necessary to connect with the hardware and collect, parse, and correct the point cloud data. This corrected data is stored globally outside of the driver in variables located in a separate class allowing it to be accessible. The final detail for developing a test product is establishing a main class that combines these steps in a basic program and implements CIP.

Table 2: Project Goals and Where They Are Implemented

Project Goal	**Program Part**
Access and parse point cloud data	Device Driver
Set up custom Toolkit API	CIP
Implement a basic interrupt routine for collisions	CIP/Main
Determine if any surrounding objects have moved	CIP
Find current position and velocity of moving objects	CIP
Calculate probability and potential severity of collisions	CIP

Table 2 above indicates the primary project goals and where they fit into the project design. Figure 15 shows how these goals progress into the final intended product. The testing for this project is fairly minimal, in order to prove that the project is functional, CIP must be implemented into an existing program with ease. If this is the case and the product is able build, compile, and function to the extent of the above goals, then the project has been proved successful. This proves that collision detection software can be simplified and improved not only for stronger collision detection, but for a greater user experience.

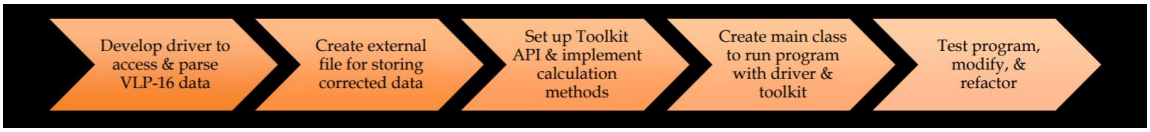


Figure 15: Project Milestone and Timeline Flowchart

4 Implementation & Methodology

This section will go through the process of developing this project – from making the VLP-16 driver to creating the CIP API to implementing everything in a test program. The interaction of the classes designed for these aspects of the project is detailed and can be seen in the class diagram below, Figure 16. The approach taken to develop this project is explained here in order of the steps taken to best encapsulate the thought process throughout development.

VLP-16 Driver Development

Creating a point cloud collision detection system requires the ability to acquire and work with point clouds for testing and other aspects of development. For this reason, the first step of this project was to make a device driver for the VLP-16. A device driver is an instance of software that connects to a piece of hardware, requests information from it, performs necessary corrections on this data, and then stores the end result. Since the purpose for the driver in this project was only to attain basic location information, the functionality of the driver need only meet these basic requirements. The VLP-16 driver functionalities and interactions are outlined in the class diagram below, 16. As this diagram shows, the driver class has methods for connecting to the device, receiving information, and modifying raw data. These three main functionalities each have many aspects necessary for achieving the main goal.

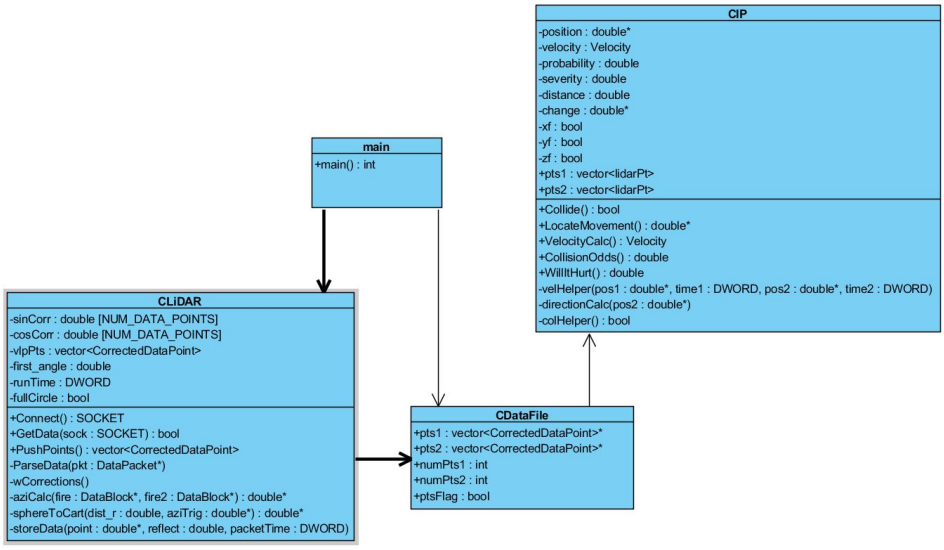


Figure 16: Class Diagram for the Program

The very first step in developing this program was to connect to the VLP-16 and receive data. In order to achieve this, the *Connect()* method was created to connect to Port 2368, which is where data from the device can be accessed [10]. This function was developed using Winsock, a Microsoft programming interface used for creating server and client applications [12]. The code for this function was based on the series of Winsock tutorials from Microsoft's online documentation, excerpts from these can be found in Appendix F. After Winsock was setup and initialized in the driver, a client socket was created to receive data from the VLP-16 which acts as a server. Following the Winsock tutorial, a client socket was created by connecting to the defined port, requesting the server's IP address, and calling various functions to check for errors [13]. Next, the socket was connected to the server and, the program was ready to receive data from the VLP-16. Since data receiving occurs later in the driver, the *Connect()* method returns the socket to be used in the *GetData()* method.

Once the connection is established, the *Connect()* method initializes some of the global variables and calls the *wCorrections()* method to set the omega corrections for each laser. Omega values, the vertical angles of the Spherical coordinate system, vary by the model of the Velodyne device and are given as a list in the user manual [10]. This list of values is used to determine the sine and cosine of each vertical angle

for later use in the conversion from Spherical to Cartesian coordinates.

The VLP-16 continuously sends data when it is connected. For this reason, a function, the *GetData()* method, was created in order to poll the device for information when needed. Polling the device regulates the amount of information processed by the driver, thus making the program more efficient. The *GetData()* method uses the previously created socket to periodically receive data from the VLP-16. This data is immediately placed in a character buffer of size 1248 – since a character is the equivalent of 1 byte, this allocates the exact number of bytes as one VLP-16 data packet which is in hexadecimal. In order to be able to access this data in the form of a data packet, a *DataPacket* struct was created, shown in Figure 17. This struct is broken down exactly as the data packets are arranged, as shown in Figure 12. To achieve this, additional structs *DataPoint, DataBlock,* and *Header* were also created, Figure 17. Since the received data is stored as a character array, the information must to be copied from this area of memory into an area of memory already allocated for a *DataPacket*. With the data properly formatted and stored in memory, the *ParseData()* method is called to handle the raw data.

```
typedef struct
{
   unsigned short distance;
   unsigned char reflectivity;
} DataPoint;

struct DataBlock
{
   unsigned short flag;
   unsigned short alpha;
   DataPoint dataPoints[NUM_DATA_POINTS];
};

struct DataPacket
{
   DataBlock dataBlocks[NUM_DATA_BLOCKS];
   unsigned int timestamp;
   unsigned char mode; // return mode
   unsigned char product; // product ID
};

struct Header
{
   BYTE head[42];
};
```

Figure 17: Structs Created for the Raw Data Packet

The data from the VLP-16 then goes through the *ParseData()* method, a function that handles the corrections for sensor placement and arranges the data in an easily manageable way. The biggest responsibility of this function is converting the data from Spherical to Cartesian coordinates. In order to do this, the omega value for each laser was stored in a global constant array and used to create arrays of sine and cosine corrections for the corresponding lasers. These values are given in VLP-16 user manual and are listed in order of the lasers.

The next important bit of information for the conversion is the alpha angle, or the azimuth, of each firing sequence. Since only one azimuth is given per data block, the azimuth for the second firing sequence needed to be interpolated in order to have a more accurate data conversion. The code for interpolating the second azimuth of each data block was based on the azimuth pseuedocode listed in both user manuals [10][14], seen in Appendix H. To attain this value, the difference between the azimuth

values of the current data block and the next data block was divided in half and added to the value of the current azimuth. These azimuth values are used to make the remaining sine and cosine functions needed for the conversion to Cartesian. The final value needed to complete this conversion is the distance from the sensor to the measured data point. With all of this information calculated and assigned to the correct lasers, the data points can be converted to Cartesian coordinates by using the formulas given in Table 1 and shown in Figure 5.

Creating the CIP Toolkit API

Since the API's functionality was essentially math and "getter" methods, a logical place to start was with what attributes the class will need to be able to provide. For this reason, CIP was essentially programmed in reverse. First, the function calls were implemented in the main function as seen in Figure 18. Next, the attributes and their respective getter methods where added to the header file and implemented in the most basic form in the cpp file. Once the attributes are established, they need to be initialized somewhere, so constructor and decontructor methods were added to the CIP class. With all of the set up complete, all that remained was adding the math. The math implemented was fairly basic – just typical Pythagorean Theorem math and change in distance and speed calculations.

Forming the Complete Program

Once both the driver and toolkit were complete, all that remained was implementing them into one program. This was accomplished by creating a main class with a *main()* function that handled function calls. With the code arranged in the main class, testing began. At first, testing ran into several challenges with correctly establishing the VLP-16 as a server, but once the connection was successful, the program was able to build, compile, and function properly.

```
// Implement Toolkit API
if (cip.Collide())
{
    cout << "Possible collision detected" << endl;

    double *pos = cip.LocateMovement();
    cout << "Current position of movement in meters from (0, 0, 0) of the sensor: ("
        << pos[0] << ", " << pos[1] << ", " << pos[2] << ")" << endl;

    Velocity vel = cip.VelocityCalc();
    cout << "Current observed velocity of movement in m/s: " << vel.speed << endl;

    double prob = cip.CollisionOdds();
    cout << "Probability of collision: " << prob << endl;

    double sever = cip.WillItHurt();
    cout << "Potential severity of collision (will it hurt?): " << sever << endl;
}
```

Figure 18: Implementing CIP in the Main Class

5 Results & Discussion

This section goes through the numerous learning experiences presented throughout project development and testing. The significant challenges encountered are described with their respective solutions and outcomes. Also, discussed in this section are some aspects of the project that have the potential to be more precise and efficient, and why this potential was not acted on given the project's specific needs.

Some of the main challenges encountered included: getting received data into a data packet, getting data into CIP, and making the API as simple to implement as possible. This last point required frequent refactoring and process rethinking to get the current solution. Due to the goals of this project, implementing an interrupt routine seemed rather impractical for achieving the desired adaptable and intuitive solution. For this reason, the *Collide()* function was created. This method can easily be used as a flag for an external interrupt routine without making the program too cumbersome or limiting possible applications. Thus, this design change broadened CIPs usage and simplified its implementation into existing products.

Testing the VLP-16 yielded an error due to setting it up properly with the computer. In order for the VLP-16 to behave as the expected server, the device running the program must be disconnected from the internet, and the IP address must be adjusted accordingly to establish the VLP-16 as the server. Without this step, the program will build and compile but will not get past the *recv()* method in the *GetData()* method.

As mentioned, the device driver created is a basic version of an incredibly complex and diverse system. If this were to be used for anything more intensive than testing, it would be advisable to implement threads and a more precise system for polling. Threads would help ensure that data will not be lost during processing. The program takes time to correct and alter data, and while this is happening, the sensor is still producing vast amounts of data. By implementing threads, much less of this data would be lost during other processes. This is especially important when working with high speed applications, since the missing data effectively reduces the system's capability to respond to rapid changes. Since the program is very small, there is not much concern that a significant amount of data will be lost due to a busy processor. In this program, the data is collected back to back in the main function and the calculations are performed afterwards, so the two readings being compared will not

be affected much by a lack of threads. This method of implementation does not fully use the polling function of the driver completely either. But, again, since the data is being compared in the way it is, this is not important for my program. Overall, for a true collision detection program, it would be imperative to have accurate data at precise times to ensure the data is correct and processed as quickly as possible.

Determining a full sweep of points, a full rotation, around the VLP-16 proved a bit challenging as well. Since the readings do not necessarily start at 0 degrees every time, the initial value must be stored, but the azimuth value can change by small or large increments and is in hundredths of a degree. For this reason, it is highly improbable that a full rotation around would end at the same starting value. Because of this, some of the data may not yield a full point cloud around the device. Given more time, it would be preferable to look into better solutions for this problem.

For the Toolkit API, much of the collision detection is very simple and would not actually be specific enough for a system or user to know how to act or the true status/threat of surrounding movement. Since the primary focus of this project was implementing CIP into an existing program, the toolkit itself is a proof of concept for developing a program complete enough for testing implementation took a significant amount of work to develop properly. Creating more extensive calculations in the toolkit would yield much more accurate tests and results for testing collision detection.

Overall, the testing of CIP in a complete program went very well. The program builds, compiles, and runs effectively without any warnings or errors. Implementing CIP into the program took less than 10 lines of code in the main function and has very few requirements to be successful. In implementing CIP, the same challenge as receiving data from the VLP-16 presented itself: CIP uses its own data struct for the vectors of information, meaning the inputted point clouds must be set to this data type. In order to accomplish this, the same solution was used as before, and the data was copied from the original memory space to a space in memory previously cast as the CIP data type. If the program CIP is being implemented in is fairly new, then this could be avoided by using this data type throughout the code, but for ease of implementation, it is absolutely not necessary. The other requirements of CIP are that the currently the point clouds must be vectors of the same size and the data type requires certain information and must be 3D. Further development would be able to ease these requirements to further simplify CIP implementation.

In testing the program as a whole, a full code analysis was run through the

compiler. This analysis resulted in two warnings about data loss when converting from one type of data to another in the device driver, shown in Figure 19. The data in question is the timestamp of the data packet. Given the circumstances, the possible data loss is not of significance. The time is already calculated down to the microsecond, so the couple of lost digits are so small that they do not change the value significantly enough to affect the basic collision detection software using it.

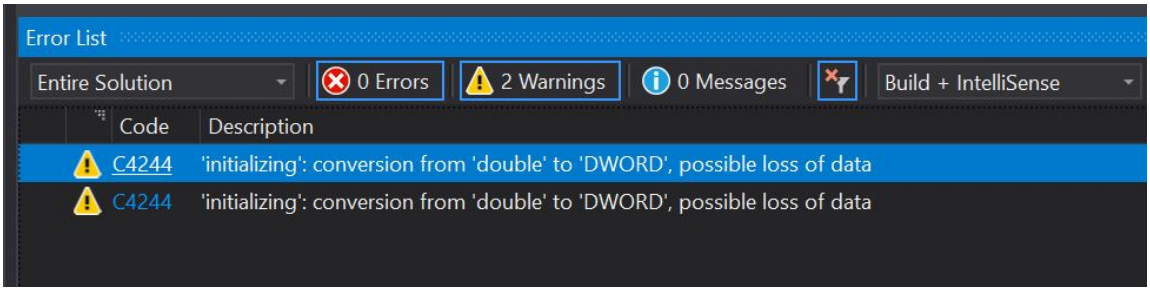


Figure 19: Results of a Code Analysis on the Program

6 Conclusions & Future Work

This project met all of its goals and succeed in creating a functional, modular, and adaptable collision detection toolkit. Due to the unique design of CIP, there are many options for pursuing the project further. Ideally, the CIP constructor will eventually take in a value to indicate the type of sensor it is receiving data from and will be able to adjust the functionality of the toolkit accordinly. For example, with the sensor used in this project, CIP could use the reflectivity value incorporated in the data point to venture into object identification. This topic alone has a countless options for further development. An ideal continuation of this project would delve deeper into the collision detection aspect in order to not only have a modular solution, but one that is also very accurate and efficient. Due to the fast pace of data collection, this would be very possible. It would even be possible to implement some sort of pathfinding for collision avoidance rather than just detection.

Developing CIP has certainly provided some unexpected twists and turns, but, all in all, the project came together nicely and succeeded in meeting every goal set. CIP has definitely become a program with the potential to continuously expand, grow, and exceed expectations. It will be exciting to see what future developer's dream up for CIP's next challenge!

References

[1] P. Lienert, "Cost of driverless vehicles to drop dramatically: Delphi ceo," *Journal Insurance*, Dec. 2017. [Online]. Available: https://www.insurancejournal.com/news/national/2017/12/05/473134.htm.

[2] J. Fingas, "Cost of driverless vehicles to drop dramatically: Delphi ceo," *Journal Insurance*, Feb. 2019. [Online]. Available: https://www.engadget.com/2019/02/03/waymo-self-driving-cars-disengagement-rate/.

[3] J. Clover, "Apple reports self-driving car disengagements to dmv, earns worst rank," *MacRumors*, Feb. 2019. [Online]. Available: https://www.macrumors.com/2019/02/12/apple-self-driving-car-disengagements-report/.

[4] J. Lee, "Adopt real-time data analytics or get left behind," *IDG Communications, Inc.*, Nov. 2017. [Online]. Available: https://www.cio.com/article/3238475/adopt-real-time-data-analytics-or-get-left-behind.html.

[5] "Module visualization documentation," *Point Cloud Library (PCL)*, [Online]. Available: http://docs.pointclouds.org/trunk/group__visualization.html.

[6] "Point cloud data," Mar. 2019. [Online]. Available: https://www.usna.edu/Users/oceano/pguth/md_help/html/pt_clouds.htm.

[7] R. Dalheim, "Terrestrial laser scanners collect forest data in awesome animation," *Woodworking Network*, Apr. 2018. [Online]. Available: https://www.woodworkingnetwork.com/video/terrestrial-laser-scanners-collect-forest-data-awesome-animation.

[8] E. W. Weisstein, "Spherical coordinates," *MathWorld–A Wolfram Web Resource*, Apr. 2019. [Online]. Available: http://mathworld.wolfram.com/SphericalCoordinates.html.

[9] D. Q. Nykamp, "Spherical coordinates," *Math Insight*, [Online]. Available: http://mathinsight.org/spherical_coordinates.

[10] *Vlp-16 user manual*, Velodyne LiDAR, Velodyne LiDAR, Inc., 2018.

[11] K. Murdy, "Hysweep: Filtering topographic laser data," *Sounding Better Newsletter*, Sep. 2018. [Online]. Available: http://www.hypack.com/about-hypack/sounding-better-newsletter/2018-archive/september-2018.

[12] J. Kennedy and M. Satran, "Getting started with winsock," May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/winsock/getting-started-with-winsock.

[13] ——, "Creating a socket for the client," May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/winsock/creating-a-socket-for-the-client.

[14] *User's manual and programming guide: Vlp-16*, Velodyne LiDAR, Velodyne LiDAR, Inc., 2016.

[15] ——, "Creating a basic winsock application," May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/winsock/creating-a-basic-winsock-application.

[16] ——, "Initializing winsock," May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/winsock/initializing-winsock.

[17] ——, "Connecting to a socket," May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/winsock/connecting-to-a-socket.

[18] ——, "Sending and receiving data on the client," May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/winsock/sending-and-receiving-data-on-the-client.

Appendix A: Main Program CPP File

```
#include "LiDAR.h"
#include "dataFile.h"

int main()
{
   CLiDAR puck;
   CIP cip;
   SOCKET sock;
   sock = puck.Connect();

   while (1)
   {
      puck.GetData(sock);
      savedData->pts1 = &puck.PushPoints();
      //Do I want this? Sleep(3); // will miss about 2 data packets
      puck.GetData(sock);
      savedData->pts2 = &puck.PushPoints();

      // Copy data from savedData to CIP vectors
      vector<lidarPt> *src1 = (vector<lidarPt>*)(savedData->pts1);
      vector<lidarPt> *src2 = (vector<lidarPt>*)(savedData->pts2);
      vector<lidarPt> *ptr1, *ptr2;
      ptr1 = &(cip.pts1); ptr2 = &(cip.pts2);

      memcpy(ptr1, src1, savedData->numPts1);
      memcpy(ptr2, src2, savedData->numPts2);

      int size1 = cip.pts1.size; int size2 = cip.pts2.size;
      int diff = size1-size2;

      // Ensure that the vectors are the same size
      for (int i = 0; i < diff; i++)
      {
         if (diff > 0)
            cip.pts2[size2 + i] = cip.pts1[size1 - (diff + i)];
```

```
        else if(diff < 0)
37          cip.pts1[size1 + i] = cip.pts2[size2 - (diff + i)];
    }
39
    // Implement Toolkit API
41  if (cip.Collide())
    {
43      cout << "Possible collision detected" << endl;

45      double *pos = cip.LocateMovement();
        cout << "Current position of movement in meters from (0, 0, 0) of
            the sensor: ("
47          << pos[0] << ", " << pos[1] << ", " << pos[2] << ")" << endl;

49      Velocity vel = cip.VelocityCalc();
        cout << "Current observed velocity of movement in m/s: " << vel.
            speed << endl;
51
        double prob = cip.CollisionOdds();
53      cout << "Probability of collision: " << prob << endl;

55      double sever = cip.WillItHurt();
        cout << "Potential severity of collision (will it hurt?): " <<
            sever << endl;
57  }
  }
59}
```

Appendix B: VLP-16 Driver CPP File

```
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include "LiDAR.h"

#define LIDAR_PORT 2368

// Vertical angles (w) in spherical coordinates
const double VertAng[] =
{ // Specific to VLP-16
   -15.0,
     1.0,
   -13.0,
     3.0,
   -11.0,
     5.0,
    -9.0,
     7.0,
    -7.0,
     9.0,
    -5.0,
    11.0,
    -3.0,
    13.0,
    -1.0,
    15.0
};

struct addrinfo *result = NULL,
   *ptr = NULL,
   hints;

#pragma comment(lib, "Ws2_32.lib")
```

```
36 CLiDAR::CLiDAR()
   {
38    sinCorr[0] = { 0. };
      cosCorr[0] = { 0. };
40    first_angle = 0;
      runTime = 0;
42    fullCircle = false;
   }
44
   CLiDAR::~CLiDAR()
46 {
   ;
48 }

50 //---------------------------------------------------------------------
   // Connect to LiDAR
52 SOCKET CLiDAR::Connect( )
   {
54    WSADATA wsaData;
      int iResult;
56
      // Initialize Winsock
58    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
      if (iResult != 0)
60    {
         printf("WSAStartup failed: %d\n", iResult);
62       return false;
      }
64
      sockaddr_in puckAddr;
66    puckAddr.sin_family = AF_INET; // PF_INET;
      puckAddr.sin_port = htons(LIDAR_PORT);
68    puckAddr.sin_addr.s_addr = htonl(INADDR_ANY);

70    SOCKET sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
      if (sock == INVALID_SOCKET)
72    {
```

```
      printf("Invalid Data Socket");
74    return false;
   }
76
   if (bind(sock, (sockaddr *)&puckAddr, sizeof(puckAddr)) == SOCKET_ERROR)
78 {
      printf("Socket bind error");
80    return false;
   }
82
   fullCircle = false;
84 first_angle = 0;
   vlpPts.clear();
86 wCorrections();

88 return sock;
}
90
//---------------------------------------------------------------------------
92 // Set omega corrections, corresponding to the respective laser number
void CLiDAR::wCorrections()
94 {
   // Set aside memory
96 memset(sinCorr, 0, sizeof(double)*NUM_DATA_POINTS);
   memset(cosCorr, 0, sizeof(double)*NUM_DATA_POINTS);
98
   // 16 beams -- a "firing" block contains 2 firings of 16
100 // sine and cosine of omega to be used in spherical to cartesian
       conversion
   for (int i = 0; i < NUM_DATA_POINTS / 2; i++)
102 {
      sinCorr[i] = sin(VertAng[i] * M_PI / 180.);
104   cosCorr[i] = cos(VertAng[i] * M_PI / 180.);
   }
106 }

108 //---------------------------------------------------------------------------
```

```
// Poll - poll device about 20 times / second for new data
// return true if data is available and false is not
bool CLiDAR::GetData(SOCKET sock)
{
   int count = 0;
   char buf[1248];
   while (count < 10)
   {
      // Receives one instance of data from the LiDAR
      // buf stores complete raw data packet
      int q = recv(sock, buf, sizeof(buf), 0);
      DataPacket *src = (DataPacket*)( ((BYTE*)buf) + sizeof(Header) );

      DataPacket pac, *ptr;
      ptr = &pac;

      // parse new topo message
      memcpy( ptr, src, sizeof(DataPacket) );
      ParseData(ptr);

      if (fullCircle)
         break;

      count++;
   }
   return (count == 0) ? false : true;
}

//-----------------------------------------------------------------------
// Handle LiDAR data
void CLiDAR::ParseData( DataPacket *pkt )
{
   DWORD time_milli = pkt->timestamp / 1000.; // Time in milliseconds past
       the hour

   // Iterate through the data blocks 0-11
   for (int i = 0; i < NUM_DATA_BLOCKS; i++)
```

```
    {
146     DataBlock *fire = (DataBlock*)&pkt->dataBlocks[i];
        if (i == 0) first_angle = fire->alpha;
148
        // Check block flag to ensure not bad data
150     if (fire->flag != 0xeeff)
            continue;
152
        DataBlock *fire2 = (DataBlock*)&pkt->dataBlocks[i + 1];
154     double *aziTrig = aziCalc(fire, fire2);

156     if(fullCircle)
            return;
158
        // Iterate through data points 0-31
160     for (int j = 0; j < NUM_DATA_POINTS; j++)
        {
162         double dist_r = fire->dataPoints[j].distance;

164         // Check distance to point & skip bad shots
            if (dist_r == 0)
166             continue;

168         if (runTime == 0)
                runTime = time_milli;
170         int laser = (j < 16) ? j : j - 16;

172         // Timing correction, relative to start of pkt
            double time_micro = (i * 110.592) + (laser * 2.304);
174         if (j >= 16)
                time_micro += 55.296;
176         DWORD packetTime = round(time_micro / 1000.);
            packetTime += time_milli;
178
            double *pt = sphereToCart(dist_r, aziTrig);
180         double reflect = fire->dataPoints[j].reflectivity;
```

```
182         storeData(pt, reflect, packetTime);
      }
184   }
}
186
//------------------------------------------------------------------------
188 // Interpolates the azimuth for the second firing sequence of a data block
double* CLiDAR::aziCalc(DataBlock *fire, DataBlock *fire2)
190 {
   // convert azimuth value from 100ths to degrees
192   double aziAngle = fire->alpha / 100.0;
   double aziAngle3 = fire2->alpha / 100.0;
194
   // Azimuth interpolation to determine missing azimuth value
196   // follows 'pseudo-code' velodyne manual pg 25 or 65 dep. on version
   double aziAngle2;
198
   if (aziAngle3 < aziAngle)
200      aziAngle3 += 360.;
   aziAngle2 = aziAngle + ((aziAngle3 - aziAngle) / 2);
202   if (aziAngle2 > 360.)
      aziAngle2 -= 360.;
204
   // Detect full rotation around device
206   if (aziAngle <= first_angle)
      fullCircle = true;
208
   // Filter by rotational angle
210   // convert azimuth/alpha to radians
   double azi = (aziAngle * M_PI / 180.);
212   double azi2 = (aziAngle2 * M_PI / 180.);
   double cos_azi = cos(azi); double sin_azi = sin(azi);
214   double cos_azi2 = cos(azi2); double sin_azi2 = sin(azi2);

216   static double trig[] = { cos_azi, sin_azi, cos_azi2, sin_azi2 };
   return trig;
218 }
```

```
220 //---------------------------------------------------------------------
    // Converts from spherical to cartesian coordinates
222 double* CLiDAR::sphereToCart(double dist_r, double *aziTrig)
    {
224    // Counter tells which data point: 0 through 31
       static int count = 0;
226
       double cos_azi, sin_azi;
228    int laser = (count < 16) ? count : count-16;

230    // Distance to point in meters--account for 2mm granularity
       dist_r *= 0.002;
232    double dist_xy = dist_r * cosCorr[laser]; // r * cos(w)

234    // First or second firing sequence in the data block?
       if (count < 16)
236    {  // use azimuth for 1st firing
          cos_azi = aziTrig[0];
238       sin_azi = aziTrig[1];
       }
240    else
       {  // use interpolated azimuth for 2nd firing
242       cos_azi = aziTrig[2];
          sin_azi = aziTrig[3];
244    }

246    // conversion results
       double x = (dist_xy * sin_azi);
248    double y = (dist_xy * cos_azi);
       double z = (dist_r * sinCorr[laser]);
250
       count++;
252
       static double pt[] = {x, y, z};
254    return pt;
    }
```

```
256
//-----------------------------------------------------------------------
258 // Stores data externally
void CLiDAR::storeData(double *point, double reflect, DWORD packetTime)
260 {
   //if (packet_time == 0) packet_time = timetag;
262
   CorrectedDataPoint pt;
264    double x = point[0], y = point[1], z = point[2];

266    pt.x = x;
   pt.y = y;
268    pt.z = -z;

270    pt.reflectivity = reflect;
   pt.timeDelay = packetTime - runTime;
272    vlpPts.push_back(pt);

274    // Approx. num pts per rotation -- assuming 600 rpm
   if (vlpPts.size() >= 30000)
276       fullCircle = true;
}
278
//-----------------------------------------------------------------------
280 // Fills a global array with the current data points
vector<CorrectedDataPoint> CLiDAR::PushPoints()
282 {
   vector<CorrectedDataPoint> points = vlpPts;
284
   fullCircle = false;
286    runTime = 0;
   vlpPts.clear();
288
   return points;
290 }
```

Appendix C: VLP-16 Driver Header File

```
#pragma once

#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <iostream>

#include "dataFile.h"

const int NUM_DATA_POINTS = 32;
const int NUM_DATA_BLOCKS = 12;

#pragma pack(push, 1)
typedef struct
{
   unsigned short distance;
   unsigned char reflectivity;
} DataPoint;

struct DataBlock
{
   unsigned short flag;
   unsigned short alpha;
   DataPoint dataPoints[NUM_DATA_POINTS];
};

struct DataPacket
{
   DataBlock dataBlocks[NUM_DATA_BLOCKS];
   unsigned int timestamp;
   unsigned char mode; // return mode
   unsigned char product; // product ID
```

```
};

struct Header
{
   BYTE head[42];
};

struct LaserCorrection
{
   double azimuth; // angle alpha in hundredths of a degree
   double verticalCorr;
   double distanceCorr;
   double verticalOffsetCorr;
   double horizontalOffsetCorr;
   double sinVertCorr;
   double cosVertCorr;
   double sinVertOffsetCorr;
   double cosVertOffsetCorr;
};

#pragma pack(pop)


class CLiDAR;
class CLiDAR
{
public:
   CLiDAR();
   ~CLiDAR();
   SOCKET Connect();
   bool GetData(SOCKET sock);
   vector<CorrectedDataPoint> PushPoints();

private:
   void ParseData(DataPacket *pkt);
   void wCorrections();
   double* aziCalc(DataBlock *fire, DataBlock *fire2);
```

```
73    double* sphereToCart(double dist_r, double *aziTrig);
      void storeData(double *point, double reflect, DWORD packetTime);
75
      double sinCorr[NUM_DATA_POINTS];
77    double cosCorr[NUM_DATA_POINTS];
      vector<CorrectedDataPoint> vlpPts;
79    double first_angle;
      DWORD runTime;
81    bool fullCircle;
};
```

Appendix D: CIP Toolkit API CPP File

```
#include "CIP.h"

//--------------------------------------------------------------------
// Constructor
CIP::CIP()
{
   position[0] = 0.0; position[1] = 0.0; position[2] = 0.0;
   velocity.speed = 0.0;
   velocity.direction = 0;
   probability = 0.0;
   severity = 0.0;
   distance = 0.0;
   change[0] = 0; change[1] = 0; change[2] = 0;
   xf = false; yf = false; zf = false;
}

//--------------------------------------------------------------------
// Deconstructor
CIP::~CIP()
{
;
}

//--------------------------------------------------------------------
// Given 2 point clouds determines if anything may collide with self
bool CIP::Collide(vector<CorrectedDataPoint> pts1, vector<
    CorrectedDataPoint> pts2)
{
   double pos1[3], pos2[3];
   DWORD time1, time2;
   for (int pt = 0; pt < (int)pts1.size(); pt++)
   {
      CorrectedDataPoint p1 = pts1[pt];
      pos1[0] = p1.x;
      pos1[1] = p1.y;
```

```
35      pos1[2] = p1.z;
        time1 = p1.timeDelay;
37      CorrectedDataPoint p2 = pts2[pt];
        pos2[0] = p2.x;
39      pos2[1] = p2.y;
        pos2[2] = p2.z;
41      time2 = p2.timeDelay;
        position = pos2;
43      velHelper(pos1, time1, pos2, time2);
        if (!colHelper())
45         return false;
        probability = (velocity.speed / distance) * velocity.direction;
47      severity = probability * velocity.speed;
        return true;
49    }
    return false;
51  }

53  //---------------------------------------------------------------------
    // Determines the position, direction, and speed of possible movement
55  void CIP::velHelper(double *pos1, DWORD time1, double *pos2, DWORD time2)
    {
57    // Distance calculations
    change[0] = pos2[0] - pos1[0];
59    change[1] = pos2[1] - pos1[1];
    change[2] = pos2[2] - pos1[2];
61

    distance = sqrt(pow(change[0], 2) + pow(change[1], 2) + pow(change[2],
        2));
63    directionCalc(pos2);
    velocity.speed = (distance / (time2 - time1));
65  }

67  //---------------------------------------------------------------------
    // Direction Calculations
69  void CIP::directionCalc(double *pos2)
    {
```

```
71    int count = 0, flag;
      double axis[] = { pos2[0], pos2[1], pos2[2] };
73
      while (count < 3)
75    {
         double ax = axis[count]; double delta = change[count];
77
         if (ax >= 0) // + value point
79       {
            // positive change means heading away
81          if (delta > 0)
               flag = false;
83          else
               flag = true;
85       }
         else // - point value
87       {
            // positive change means heading towards
89          if (delta > 0)
               flag = true;
91          else flag = false;
         }
93       if (flag) // if heading towards add value to direction
            velocity.direction++;
95
         // Set proper flag
97       if (count == 0) xf = flag;
         if (count == 1) yf = flag;
99       if (count == 2) zf = flag;
         count++;
101   }
}
103
//-----------------------------------------------------------------------
105 // Helper to Collide() -- determines if changes are significant
bool CIP::colHelper()
107 {
```

```
   // 2 flags and 2 meters difference in each of them
   if (velocity.direction > 1)
   {
      if (change[0] > 0.01 || change[1] > 0.01 || change[2] > 0.01)
         return true;
   }
   return false;
}

//---------------------------------------------------------------------
// Returns the location where possible collision movement was detected
double* CIP::LocateMovement()
{
   return position;
}

//---------------------------------------------------------------------
// Returns the determined velocity of the moving points
Velocity CIP::VelocityCalc()
{
   return velocity;
}

//---------------------------------------------------------------------
// Returns the probability the detected movement will result in a
    collision
double CIP::CollisionOdds()
{
   return probability;
}

//---------------------------------------------------------------------
// Returns potential severity of collision
double CIP::WillItHurt()
{
   return severity;
}
```

Appendix E: CIP Toolkit API Header File

```
#pragma once

using namespace std;

#include <vector>
#include <windows.h>
#include <math.h>
#include "dataFile.h"

struct lidarPt
{
   double x;
   double y;
   double z;
   double reflectivity;
   DWORD timeDelay;
};

struct Velocity
{
   double speed;
   int direction;
};

class CIP;
class CIP
{
public:
   CIP();
   ~CIP();

   bool Collide(vector<CorrectedDataPoint> pts1, vector<CorrectedDataPoint>
       pts2);
   double *LocateMovement();
   Velocity VelocityCalc();
```

```
    double CollisionOdds();
    double WillItHurt();

private:
    void velHelper(double *pos1, DWORD time1, double *pos2, DWORD time2);
    void directionCalc(double *pos2);
    bool colHelper();

    double *position;
    Velocity velocity;
    double probability;
    double severity;
    double distance;
    double *change;
    bool xf, yf, zf;
};
```

Appendix F: Global Data Settings Files

Header File

```
#pragma once

using namespace std;

#include <windows.h>
#include <vector>

struct CorrectedDataPoint
{
   double x;
   double y;
   double z;
   double reflectivity;
   DWORD timeDelay;
};

class CDataFile
{
public:
   vector<CorrectedDataPoint> pts1;
   vector<CorrectedDataPoint> pts2;
};

// global saved data
extern CDataFile *savedData;
```

CPP File

```
#include "dataFile.h"

// global settings
CDataFile *savedData;
```

Appendix G: Winsock Tutorial Code

Winsock Application Setup [15]

```
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

int main() {
  return 0;
}
```

Winsock Initialization [16]

```
int iResult;

// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != 0) {
    printf("WSAStartup failed: %d\n", iResult);
    return 1;
}
```

Using Winsock to Create a Client Socket [13]

```
struct addrinfo *result = NULL,
                *ptr = NULL,
                hints;

ZeroMemory( &hints, sizeof(hints) );
hints.ai_family = AF_UNSPEC;
```

```
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
```

```
#define DEFAULT_PORT "27015"

// Resolve the server address and port
iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo failed: %d\n", iResult);
    WSACleanup();
    return 1;
}
```

```
// Attempt to connect to the first address returned by
// the call to getaddrinfo
ptr=result;

// Create a SOCKET for connecting to server
ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
    ptr->ai_protocol);
```

```
if (ConnectSocket == INVALID_SOCKET) {
    printf("Error at socket(): %ld\n", WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
    return 1;
}
```

Connecting to a Socket [17]

```
// Connect to server.
iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
if (iResult == SOCKET_ERROR) {
    closesocket(ConnectSocket);
    ConnectSocket = INVALID_SOCKET;
}
```

```
// Should really try the next address returned by getaddrinfo
// if the connect call failed
// But for this simple example we just free the resources
// returned by getaddrinfo and print an error message

freeaddrinfo(result);

if (ConnectSocket == INVALID_SOCKET) {
    printf("Unable to connect to server!\n");
    WSACleanup();
    return 1;
}
```

Sending and Receiving Data on the Client [18]

```
#define DEFAULT_BUFLEN 512

int recvbuflen = DEFAULT_BUFLEN;

char *sendbuf = "this is a test";
char recvbuf[DEFAULT_BUFLEN];

int iResult;

// Send an initial buffer
iResult = send(ConnectSocket, sendbuf, (int) strlen(sendbuf), 0);
if (iResult == SOCKET_ERROR) {
    printf("send failed: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

printf("Bytes Sent: %ld\n", iResult);

// shutdown the connection for sending since no more data will be sent
// the client can still use the ConnectSocket for receiving data
iResult = shutdown(ConnectSocket, SD_SEND);
```

```
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

// Receive data until the server closes the connection
do {
    iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
    if (iResult > 0)
        printf("Bytes received: %d\n", iResult);
    else if (iResult == 0)
        printf("Connection closed\n");
    else
        printf("recv failed: %d\n", WSAGetLastError());
} while (iResult > 0);
```

Appendix I: VLP-16 Azimuth Pseudocode

2016 User Manual [14]

```
// First, adjust for a rollover from 359.99 to 0
If (Azimuth[3] < Azimuth[1])
    Then Azimuth[3]:= Azimuth[3]+360;
Endif;
// Perform the interpolation
    Azimuth[2]:=Azimuth[1]+((Azimuth[3]-Azimuth[1])/2);
// Correct for any rollover over from 359.99 to 0
If (Azimuth[2]>360)
    Then Azimuth[2]:= Azimuth[2]-360;
Endif
```

2018 User Manual [10]

The pseudo code below illustrates the concept. K represents an index to a data point in the Nth data block, where its valid range is 0 to 31. Do this for each data block.

```
// First, adjust for an Azimuth rollover from 359.99 to 0
If (Azimuth[datablock_n+1] < Azimuth[datablock_n])
Then
    Azimuth[datablock_n+1] := Azimuth[datablock_n+1] + 360;
Endif;

// Determine the Azimuth Gap between data blocks
AzimuthGap = Azimuth[datablock_n+1] - Azimuth[datablock_n];

// Perform the interpolation using the timing firing
K = 0;
While (K < 31)
    // Determine if youre in the first or second firing sequence of the
       data block
    if (K < 16)
    Then
        // Interpolate
        Precision_Azimuth[K] := Azimuth[datablock_n] + (AzimuthGap * 2.304
```

```
            s * K) / 55.296 s);
    Else
        // Interpolate
        Precision_Azimuth[K] := Azimuth[datablock_n] + (AzimuthGap * 2.304
            s * ((K-16) + 55.296 s)) / (2 * 55.296 s);
    Endif

    // Adjust for any rollover
    If Precision_Azimuth[K] >= 360
    Then
        Precision_Azimuth[K] := Precision_Azimuth[K]  360;
    Endif
    K++;
End While
```